PYTHON ALGORITHMS · 그림으로 쉽게 설명하는

CHAPTER 01

알고리즘과 복잡도 분석

문제 해결 절차부터
로그와 복잡도 증명까지

Algorithms & Complexity Analysis

목차

CONTENTS

학습 목표

LEARNING OBJECTIVES

알고리즘의 정의와 필요성
왜 알고리즘이 필요한지 이해합니다.

알고리즘 표현 방법
자연어·순서도·유사코드·프로그래밍 언어로 표현합니다.

시간 · 공간 복잡도
실행 시간과 메모리 사용량 개념을 익힙니다.

빅오(Big-O) 표기법
성장률을 표현하고 간단한 부등식으로 증명합니다.

효율적인 알고리즘 설계
더 빠르고 좋은 알고리즘을 설계하는 감각을 기릅니다.

CHAPTER 01

이 장을 마치면
Big-O로 알고리즘의
효율을 말할 수 있어요.

SECTION 1.1

알고리즘이란?

1.1

문제(problem)와 알고리즘(algorithm)이 무엇인지, 그리고 좋은 알고리즘이 갖춰야 할 조건을 알아봅니다.

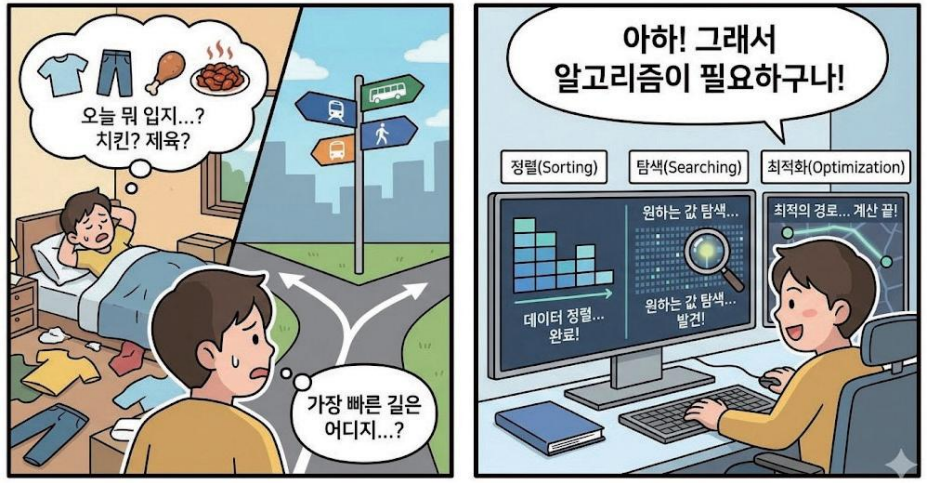
문제(Problem)란?

일상 생활 속의 문제

"무엇을 먼저 입을까?"
"가장 빠른 길은?"
"치킨이냐, 제육이냐?"

컴퓨터 공학의 문제

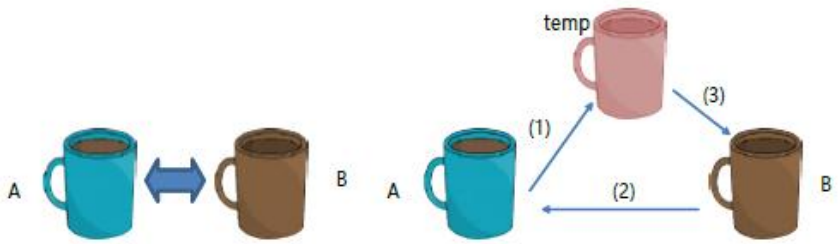
정렬 (Sorting) 데이터를 일정한 규칙으로 배열
탐색 (Searching) 원하는 값을 찾아내기
최적화 (Optimization) 조건을 만족하는 최적해 구하기



예제 1.1

일상 속 문제 — 컵 교환

A컵의 물과 B컵의 주스를 서로 바꾸려면, 임시 컵(temp)이 하나 더 필요합니다.



1. 임시 컵(temp)에 A의 물을 붓는다
2. A에 B의 주스를 붓는다
3. B에 임시 컵(temp)의 물을 붓는다

컵 교환을 변수의 값으로 따라가기

대입은 오른쪽 값을 왼쪽 변수에 복사하는 연산입니다.

실행한 명령	A	B	temp
시작	물	주스	비어 있음
temp = A	물	주스	물
A = B	주스	주스	물
B = temp	주스	물	물

원래 A의 값을 temp에 보관했기 때문에 두 값을 잃지 않고 바꿉니다.

예제 1.2

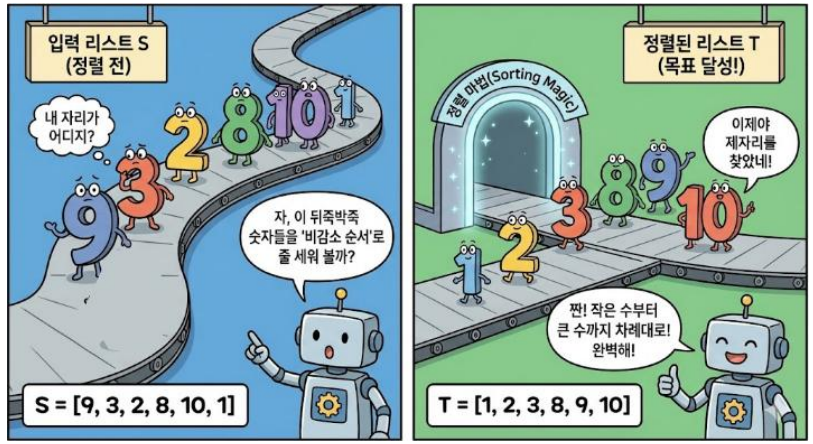
정렬 문제 (Sorting Problem)

리스트 S 안의 n개의 숫자를 비감소(nondecreasing) 순서로 정렬하라.

입력 S	9	3	2	8	10	1

정답 T	1	2	3	8	9	10

리스트의 요소들은 순서를 가지고 있습니다.



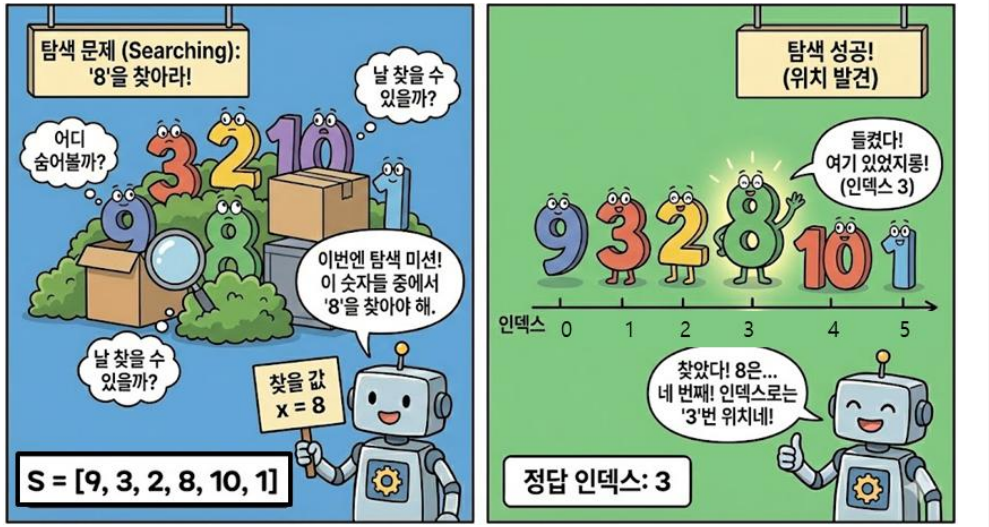
예제 1.3

탐색 문제 (Searching Problem)

리스트 S 안의 n개의 숫자 중에서 특정한 값 x를 찾아야 합니다. 정답은 값 x가 리스트의 몇 번째 위치(index)에 있는지를 나타냅니다.

탐색 S	9	3	2	8	10	1

x = 8 → 네 번째 위치, 인덱스는 **3** (0부터 시작)



알고리즘이란 무엇인가?

사람은 리스트 **[9, 3, 2, 8, 10, 1]**을 한눈에 훑어보고 순서를 떠올려 바로 정렬할 수 있습니다.

하지만 10,000개라면?
머릿속에서 한눈에 정렬하는 것은 불가능해집니다.

그래서 필요한 것
허용된 모든 입력에 대해 정답을 내고 종료하는,
명확한 단계별 절차입니다.

알고리즘 1.1

순차 탐색 알고리즘

예제 1.3의 탐색 문제는 다음과 같이 파이썬으로 작성할 수 있습니다.

linear_search.py

```python
# 문제: 리스트 S 안에서 target을 찾는다.
# 입력: 숫자들이 저장된 리스트
# 출력: S 안에서 target의 위치(없으면 -1)
def linear_search(S, target):
    for i in range(len(S)):      # 처음부터 끝까지 순서대로 탐색
        if S[i] == target:       # 값이 일치하면
            return i             # 해당 위치(인덱스) 반환
    return -1                    # 찾지 못하면 -1 반환
```

순차 탐색의 실행 기록

S = [9, 3, 2, 8, 10, 1], target = 8, 인덱스는 0부터 셉니다.

확인 횟수	인덱스 i	읽은 값 S[i]	8과 같은가?	다음 동작
1	0	9	거짓	계속
2	1	3	거짓	계속
3	2	2	거짓	계속
4	**3**	**8**	**참**	**인덱스 3 반환**

반환값 3은 인덱스이고, 비교 횟수는 4입니다.

알고리즘의 조건

입력 (Input)
외부에서 제공되는 0개 이상의 데이터. 입력이 없을 수도 있습니다.

출력 (Output)
최소한 하나 이상의 결과가 있어야 합니다. (결과가 없으면 알고리즘이 아님)

명백성 (Clarity)
각 명령어는 모호하지 않고 명확해야 합니다. ("적당히"는 없음)

유한성 (Finiteness)
유한한 명령 실행 후 반드시 종료되어야 합니다. (무한 루프는 알고리즘이 아님)

유효성 (Effectiveness)
각 명령은 실제로 수행 가능한 기본 연산이어야 합니다. (0으로 나누기는 불가)

알고리즘의 역사

알고리즘은 컴퓨터보다 훨씬 오래된, 인간이 오랫동안 사용해온 문제 해결 방법입니다.

기원전 300년경 · 유클리드
두 수의 최대공약수를 구하는 '유클리드 알고리즘'

기원전 200년경 · 에라토스테네스
소수를 찾는 '에라토스테네스의 체'

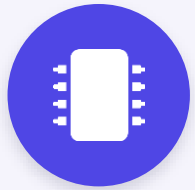
알고리즘을 공부해야 하는 이유

효율적인 프로그램 개발

정렬·탐색·데이터 처리 등 핵심 알고리즘을 알면 더 좋은 프로그램을 만들 수 있습니다.

분석적 사고력

알고리즘은 특정 문제에 대한 특별한 해결 방법 — 논리적·분석적 사고를 키웁니다.



컴퓨터 과학의 핵심

알고리즘은 컴퓨터 과학의 핵심 주제이며, 여러 분야의 데이터 처리와 문제 해결에 쓰입니다.

SECTION 1.2

알고리즘의 기술

1.2

같은 알고리즘도 자연어, 순서도, 유사코드, 프로그래밍 언어 등 다양한 방법으로 표현할 수 있습니다.

알고리즘을 기술하는 방법

자연어

영어·국어 같은
일상 언어

순서도

flowchart로
흐름을 시각화

유사 코드

pseudo-code로
핵심 논리 표현

예제 문제 크기순으로 정렬되지 않은 숫자 리스트가 있을 때, 그 중에서 **최대값을 찾는** 것이 우리가 해결할 문제입니다.

자연어로 기술하는 방법

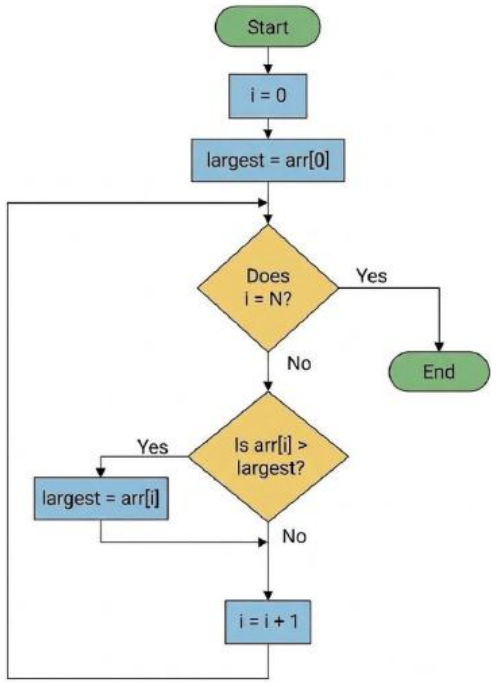
자연어는 아이디어를 표현하거나 절차를 구체화하는 초기 단계에 자주 쓰이지만, 단어가 모호하거나 여러 의미로 해석될 수 있습니다.

① 리스트의 첫 번째 숫자가 가장 크다고 가정한다.

② 남아있는 숫자들을 하나씩 조사하여, 현재의 최대값보다 크면 노트에 적는다.

③ **모든 숫자를 조사한 후, 노트에 가장 나중에 적힌 숫자가 최대값이 된다.**

순서도로 기술하는 방법



기호	의미
→	화살표는 알고리즘이 진행하는 방향을 나타낸다.
⬭	수행의 시작(start), 종료(end)
▭	처리(process)를 나타낸다. 예를 들어서 변수 x에 1을 더하는 연산이 여기에 해당된다.
◇	판단(decision)을 나타낸다. yes/no 질문이나 true/false 검사가 여기에 해당된다. 일반적으로 이 도형에서 나가는 2개의 화살표가 있다.
▱	입력(input)이나 출력(output)을 나타낸다. 예를 들어서 사용자로부터 정수를 받아서 변수 x에 저장하는 연산이 여기에 해당된다.

순서도(flowchart)는 알고리즘의 작업 순서와 논리의 흐름을 도형으로 시각화한 그림입니다.

한눈에 파악
복잡한 논리를 한눈에 파악할 수 있습니다.

오류를 쉽게 발견
설계 단계에서 오류를 쉽게 찾아냅니다.

유사코드로 기술하는 방법

유사 코드(pseudocode)는 자연어와 실제 코드의 중간 형태로, 특정 언어 문법에 구애받지 않고 핵심 논리만 간결하게 표현합니다.

find_largest.pseudo

```
// 문제: 리스트 arr 안에서 가장 큰 값을 찾는다.
// 입력: n개의 숫자가 저장된 리스트 arr
// 출력: 리스트 arr의 최대값
largest ← arr[0]
for i ← 1 to n-1 do
    if arr[i] > largest then
        largest ← arr[i]
return largest          // 리스트의 최대값 반환
```

프로그래밍 언어로 기술하는 방법

구현할 언어의 문법대로 직접 코드로 작성하면, 실제 프로그램이 될 수 있어 구현에 가장 가깝습니다.

get_largest.py

```
# 문제: 리스트 arr 안에서 가장 큰 값을 찾는다.
# 입력: n개의 숫자가 저장된 리스트 arr
# 출력: 리스트 arr의 최대값
def get_largest(arr):
    largest = arr[0]                # 첫 번째 값을 초기 최대값으로 설정
    for i in range(1, len(arr)):    # 두 번째 원소부터 끝까지 검사
        if arr[i] > largest:        # 현재 값이 기존 최대값보다 크면
            largest = arr[i]        # 최대값 갱신
    return largest                  # 최대값 반환
```

최댓값 찾기의 상태 변화

입력 [9, 3, 2, 8, 10, 1]에서 largest가 어떻게 바뀌는지 봅니다.

단계	읽은 값	비교 전 최댓값	비교 후 최댓값
초기화	9	없음	9
i = 1, 2, 3	3, 2, 8	9	9
i = 4	10	9	**10**
i = 5	1	10	10

값 비교는 n−1회이고, 후보를 실제로 갱신하는 횟수는 입력에 따라 다릅니다.

최댓값 알고리즘의 정확성 증명

반복문 불변식: i번째 원소까지 처리하면 largest는 A[0..i]의 최댓값입니다.

01 초기화: 첫 원소만 처리했으므로 largest = A[0]가 맞습니다.

02 유지: 새 원소 A[i]와 기존 최댓값 중 큰 값을 남깁니다.
따라서 A[0..i] 전체의 최댓값을 보관합니다.

03 종료: 마지막 원소까지 처리하면 전체 배열의 최댓값입니다.

04 종료 보장: i가 1씩 커지고, 남은 원소 수가 1씩 줄어듭니다.

SECTION 1.3

동일한 문제에 대한 여러 알고리즘

1.3

같은 문제라도 서로 다른 아이디어의 알고리즘이 존재하며, 그 속도는 크게 차이 날 수 있습니다.

하나의 문제, 여러 개의 알고리즘

같은 문제를 해결하는 알고리즘은 서로 매우 다른 아이디어를 기반으로 할 수 있으며, 해결 속도도 크게 차이 날 수 있습니다. 최대 공약수(GCD) 문제를 예로 살펴봅시다.

완전 탐색 방법

가능한 모든 약수를 큰 값부터 차례로 검사하는 브루트 포스 (brute force) 방식

유클리드 알고리즘

나머지를 반복적으로 계산하는, 가장 오래되고 효율적인 방식

알고리즘

완전 탐색 (Brute Force)

gcd_brute_force.py

```python
# 두 정수 a, b의 최대공약수(GCD)를 구한다.
def gcd_brute_force(a, b):
    t = min(a, b)          # 두 수 중 작은 값을 t로 설정
    while t > 0:
        if a % t == 0 and b % t == 0:   # 둘 다 나누어떨어지면
            return t       # 최대공약수 반환
        t -= 1             # t를 1 감소시키며 다음 값 검사
    return 1
```

최악의 경우

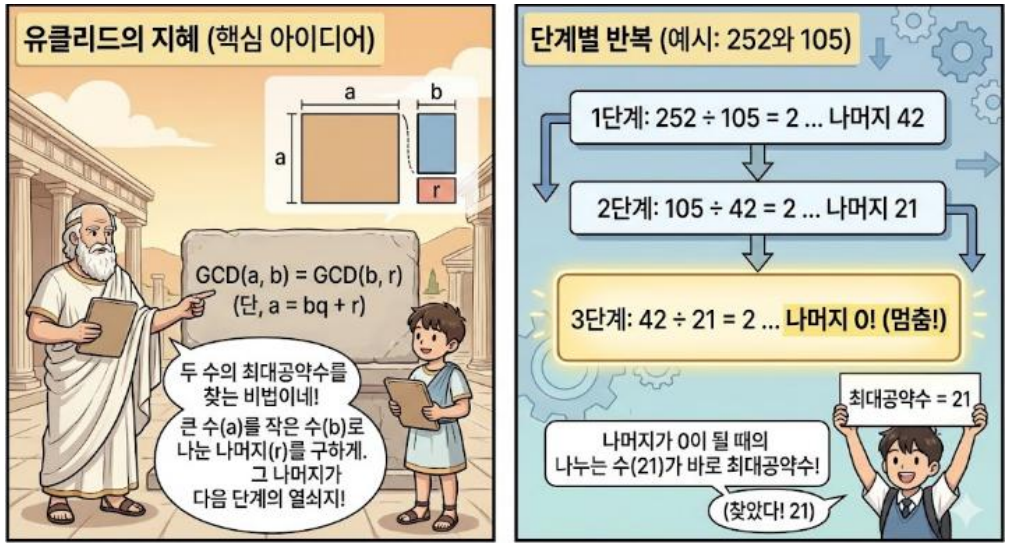
a = 1,000,000
b = 999,999

↓

999,999회

반복할 수도 있음

유클리드 알고리즘 (Euclidean Algorithm)



유클리드 알고리즘은 두 수의 최대공약수를 구하는 가장 오래되고 효율적인 알고리즘 중 하나입니다.

핵심 아이디어

큰 수를 작은 수로 나눈 나머지와 작은 수의 최대공약수는, 원래 두 수의 최대공약수와 같다.

유클리드 알고리즘의 나머지 계산

예: gcd(252, 105). 나머지가 0이 되면 마지막 0이 아닌 수가 답입니다.

반복	현재 값 (a, b)	몫과 나머지 계산	다음 값 (a, b)
1	(252, 105)	252 = 2×105 + 42	(105, 42)
2	(105, 42)	105 = 2×42 + 21	(42, 21)
3	(42, 21)	42 = 2×21 + 0	**(21, 0)**

gcd(252,105) = gcd(105,42) = gcd(42,21) = 21

나머지로 바꿔도 최대공약수가 같은 이유

a = qb + r일 때, (a,b)와 (b,r)는 공약수의 집합이 같습니다.

01 a와 b를 나누는 수 d는 r = a−qb도 나눕니다.
따라서 (a,b)의 모든 공약수는 (b,r)의 공약수입니다.

02 b와 r을 나누는 수 d는 a = qb+r도 나눕니다.
따라서 반대 방향도 성립합니다.

03 공약수가 같으므로 gcd(a,b) = gcd(b,r)입니다.

04 나머지는 0 이상 b 미만이므로 반복하면 반드시 종료합니다.

유클리드 알고리즘 — 구현

"두 수의 최대공약수는, 큰 수를 작은 수로 나눈 나머지와 작은 수의 최대공약수와 같다."

gcd_euclidean.py

```python
# 두 정수 a, b의 최대공약수(GCD)를 구한다.
def gcd_euclidean_iterative(a, b):
    while b != 0:
        temp = a          # 기존 a 값을 저장
        a = b             # a를 b로 업데이트
        b = temp % b      # b를 a % b로 업데이트
    return a
```

두 GCD 알고리즘의 반복 횟수

a = 1,000,000, b = 999,999. 여기서는 반복문 본체 실행 횟수를 셉니다.

방법	실행 과정	반복 횟수
완전 탐색	999,999부터 1까지 후보를 하나씩 확인	999,999
유클리드	나머지 1을 얻고, 다음 나머지 0에서 종료	**2**
반복 횟수 비율	999,999 ÷ 2	499,999.5

이 예에서는 반복 횟수가 약 50만 배 차이 납니다. 실제 시간 비율과는 다릅니다.

유클리드 알고리즘의 반복 수가 로그인 이유

$a \geq b > 0$일 때, 두 번의 나머지 계산 안에 작은 수가 절반 이하가 됩니다.

01 첫 나머지를 $r = a \bmod b$라고 둡니다. $0 \leq r < b$입니다.

02 $r \leq b/2$이면 이미 작은 수가 절반 이하로 줄었습니다.

03 $r > b/2$이면 다음 나머지는 $b \bmod r = b - r < b/2$입니다.

04 따라서 두 번 이내에 절반으로 줄어듭니다.
$b \geq 2$일 때 나머지 계산 횟수는 $O(\log b)$입니다.

SECTION 1.4

효율적인 알고리즘의 중요성

1.4

컴퓨터가 아무리 빨라져도 효율성은 여전히 핵심입니다. 순차 탐색과 이진 탐색으로 그 차이를 느껴봅니다.

효율적인 알고리즘의 중요성

"컴퓨터가 과거보다 훨씬 빨라졌으니, 어떤 알고리즘을 쓰든 상관없지 않을까?"

성능이 아무리 좋아지고 메모리가 싸져도, **효율성(efficiency)**은 여전히 알고리즘의 핵심입니다.

직접 느껴봅시다
전화번호부에서 친구의 이름을 찾는 두 가지 방법을 비교하며 효율성의 차이를 확인해 봅니다.

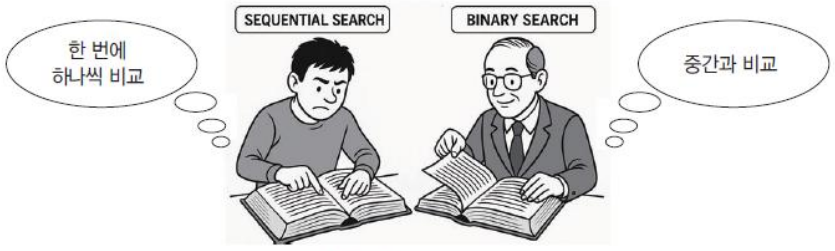
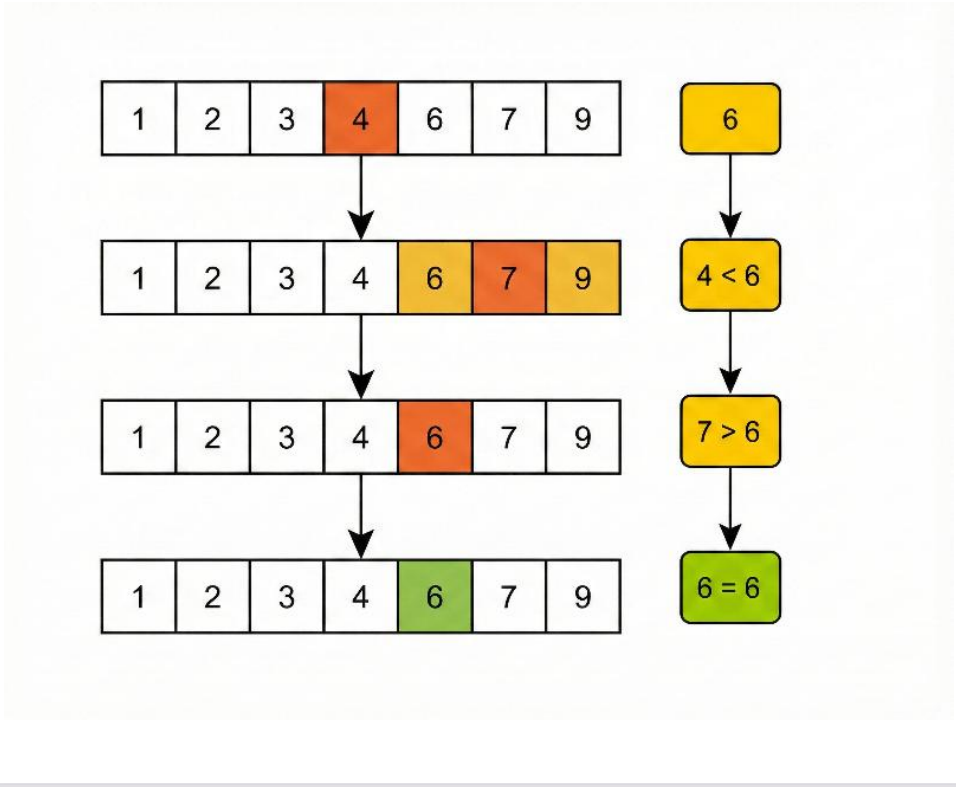


그림 1.2 알고리즘의 예

순차 탐색 vs 이진 탐색



순차 탐색 (Sequential Search)

맨 첫 페이지부터 한 줄씩 차근차근 내려가며 찾는 가장 단순한 방법.

이진 탐색 (Binary Search)

정렬되어 있음을 이용해 중간을 펼쳐 보고, 앞쪽·뒤쪽 절반으로 범위를 계속 좁혀가는 방법.

이진 탐색으로 14 찾기

정렬된 배열에서 중앙 값을 확인하고, 찾는 값이 있을 쪽만 남깁니다.

인덱스	0	1	2	3	4	5	6	7
배열의 값	2	4	6	8	10	12	**14**	16

확인 순서	탐색 인덱스 low ~ high	중앙 인덱스 mid	중앙 값 A[mid]	비교 결과와 다음 동작
1	0 ~ 7	3	**8**	14 > 8 오른쪽으로, low = 4
2	4 ~ 7	5	**12**	14 > 12 오른쪽으로, low = 6
3	**6 ~ 7**	**6**	**14**	**14 = 14, 찾음 인덱스 6 반환**

찾은 값은 14, 반환하는 인덱스는 6, 확인한 횟수는 3회입니다.

절반을 버려도 정답을 놓치지 않는 이유

불변식: 찾는 값이 존재한다면 현재 구간 [low, high] 안에 있습니다.

01 처음에는 배열 전체가 후보이므로 불변식이 성립합니다.

02 A[mid] < x이면 왼쪽 값들은 모두 x보다 작습니다.
low = mid+1로 바꿔도 정답을 잃지 않습니다.

03 A[mid] > x이면 오른쪽 값들은 모두 x보다 큽니다.
high = mid−1로 바꿔도 정답을 잃지 않습니다.

04 같으면 정답을 반환합니다. low > high이면 후보가 없으므로 −1입니다.

이진 탐색이 필요한 조건

정렬 여부와 자료구조까지 확인해야 전체 비용을 비교할 수 있습니다.

상황	가능한 선택	전체 비용
정렬된 배열, 1회 탐색	이진 탐색	O(log n)
정렬 안 된 배열, 1회	순차 탐색	O(n)
정렬 안 된 배열, q회	매번 순차 탐색	O(qn)
한 번 정렬 후 q회	비교 정렬 + 이진 탐색	O(n log n + q log n)

배열 중앙에 O(1) 시간에 접근할 수 있다는 가정도 필요합니다.

알고리즘

이진 탐색 알고리즘

binary_search.py

```python
# 정렬된 리스트 S 안에서 x를 찾는다.
def binary_search(S, x):
    low = 0
    high = len(S) - 1
    while low <= high:
        mid = (low + high) // 2   # 가운데 위치 계산
        if S[mid] == x:
            return mid            # 찾으면 인덱스 반환
        elif S[mid] < x:
            low = mid + 1         # 오른쪽 절반 탐색
        else:
            high = mid - 1        # 왼쪽 절반 탐색
    return -1                     # 못 찾으면 -1 반환
```

$\log_2 n$을 읽는 법

"밑이 2인 n의 로그"라고 읽습니다. $\log_2 n$ 전체가 하나의 값입니다.

$$2^k = n \quad \Leftrightarrow \quad \log_2 n = k$$

01 $2^3 = 8$이므로 $\log_2 8 = 3$입니다.

02 $2^5 = 32$이므로 $\log_2 32 = 5$입니다.

03 $2^0 = 1$이므로 $\log_2 1 = 0$입니다.

거듭제곱 표를 거꾸로 읽으면 로그

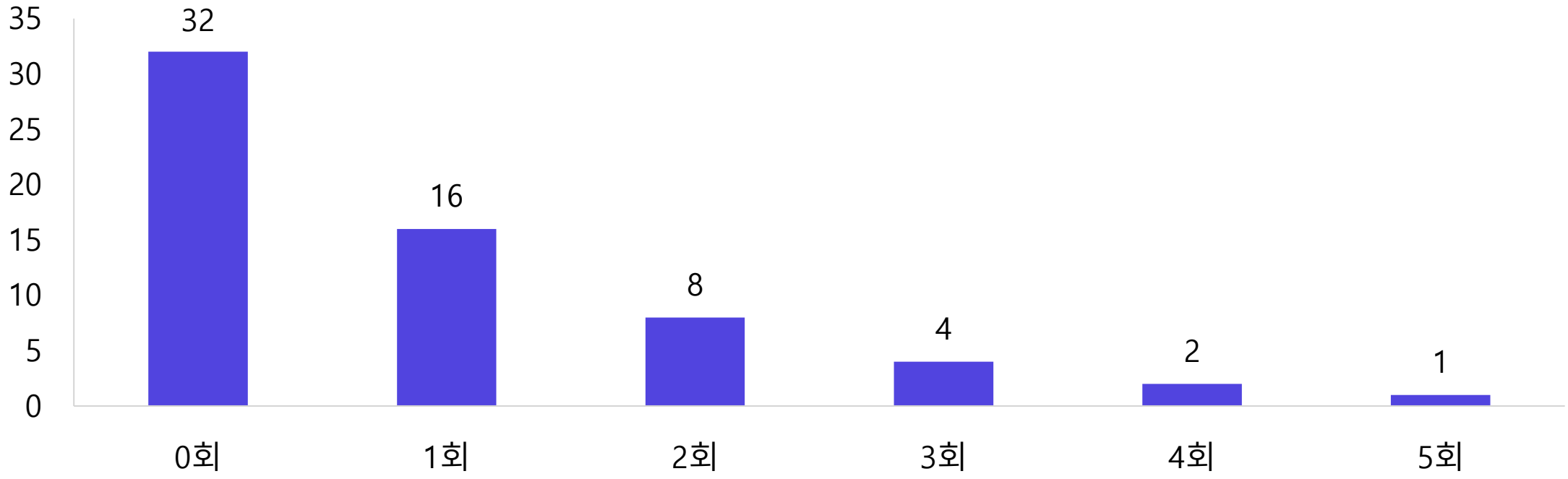
2를 한 번 더 곱할 때마다 수는 두 배, 로그 값은 1만 증가합니다.

k	0	1	2	3	4	5	10
2^k	1	2	4	8	16	32	1024
$\log_2(2^k)$	0	1	2	3	4	5	10

$\log_2(2n) = \log_2 n + 1$: 입력이 두 배가 되어도 필요한 단계는 약 1회 증가합니다.

32개를 절반씩 줄이는 그림

한 개가 남을 때까지 나눈 횟수는 5회입니다. 시작 상태는 0회입니다.



$32 \div 2^5 = 1$이므로, $5 = \log_2 32$입니다.

반복 횟수 k를 식으로 구하기

n이 2의 거듭제곱이고 매번 정확히 반으로 줄어드는 모형입니다.

01 k번 나눈 뒤의 크기: $n / 2^k$

02 한 개가 남는 조건: $n / 2^k = 1$

03 양변에 2^k를 곱하면: $n = 2^k$

04 로그의 정의를 적용하면: $k = \log_2 n$

외워야 할 연결: 절반씩 감소하는 크기와 2^k가 서로 대응합니다.

n이 2의 거듭제곱이 아닐 때

$\log_2 10$은 3과 4 사이입니다. 반복 횟수는 정수여야 합니다.

$$8 < 10 < 16 \quad \Rightarrow \quad 3 < \log_2 10 < 4$$

기호	뜻	예
[x] 바닥	x 이하의 가장 큰 정수	[3.322] = 3
[x] 천장	x 이상의 가장 작은 정수	[3.322] = 4

실수 나누기 모형에서 $n/2^k \le 1$이 되는 최소 횟수는 $[\log_2 n]$입니다.

최대 검사 횟수는 왜 $\lfloor \log_2 n \rfloor + 1$일까

$n \geq 1$, 앞의 코드에서 중앙 원소 검사 횟수의 최댓값을 셉니다.

최대 검사 횟수 = $\lfloor \log_2 n \rfloor + 1$

원소 수 n	1	7	8	10	32	1,000,000
최대 검사 횟수	1	3	4	4	6	20

한 원소가 남아도 검사해야 합니다. n=0이면 검사 0회이며 $\log_2 0$은 쓰지 않습니다.

최대 검사 횟수의 증명

C(0)=0, C(n)=1+C(⌊n/2⌋). 매번 더 큰 쪽 구간으로 진행하는 최악의 경우입니다.

01 중앙 1개를 확인한 후 남는 큰 쪽은 ⌊n/2⌋개입니다.

02 k회 뒤 큰 쪽의 크기는 $\lfloor n/2^k \rfloor$입니다.

03 후보가 0개가 되는 조건은 $n/2^k < 1$, 즉 $2^k > n$입니다.

04 이를 만족하는 최소 정수는 $k = \lfloor \log_2 n \rfloor + 1$입니다.

$n \geq 2$이면 $\log_2 n \leq C(n) \leq 2\log_2 n$이므로 $C(n) = \Theta(\log n)$입니다.

로그의 밑을 생략하는 이유

밑 a와 b가 1보다 큰 고정 상수이면 로그끼리는 상수 배 차이입니다.

$$\log_a n = \log_b n / \log_b a$$

01 정확한 계산: $\log_2 1024 = 10$, $\log_{10} 1024 \approx 3.01$

02 성장률 비교: 두 값은 고정된 상수 배 관계입니다.

03 복잡도에서는 $O(\log_2 n)$을 보통 $O(\log n)$으로 씁니다.

로그가 나오는 반복문

매번 1을 더하는지, 2를 곱하는지가 반복 횟수를 결정합니다.

```
i = 1
while i < n:
    i = i * 2
```

n = 10일 때 검사 시작 값

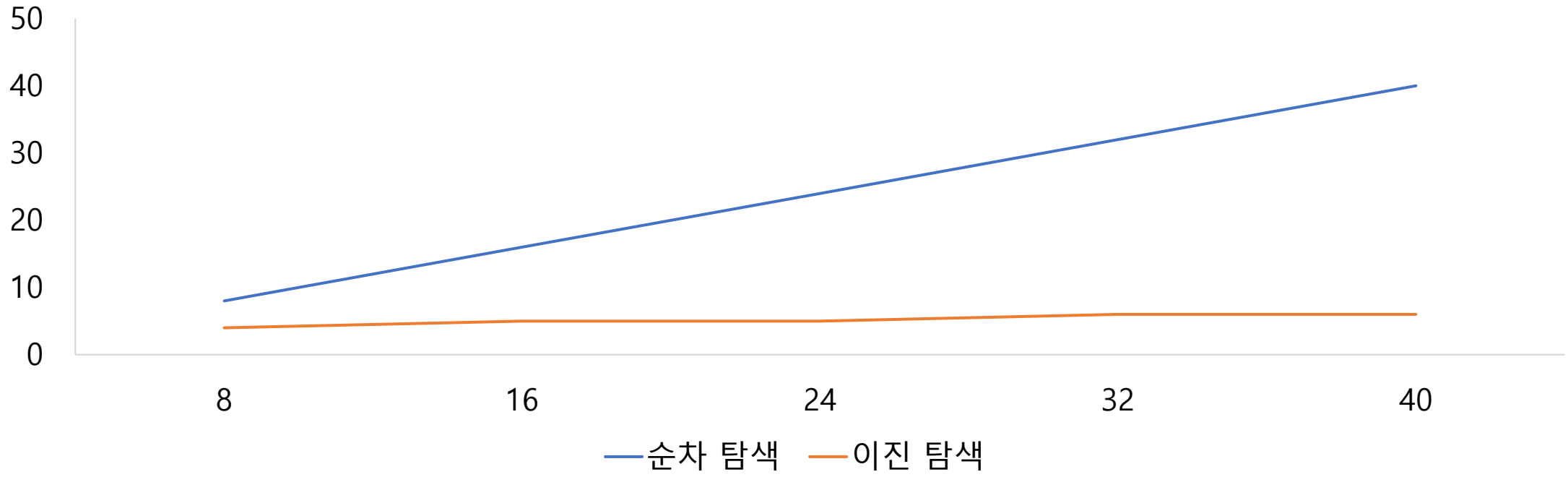
1, 2, 4, 8

16이 되면 종료

$2^k \geq n$이 되는 최소 k이므로 본체는 $\lceil \log_2 n \rceil$회 실행됩니다.

순차 탐색과 이진 탐색의 증가 비교

세로축은 최악의 검사 횟수, 가로축은 입력 원소 수입니다.



입력이 8에서 40으로 5배 늘어도, 이진 탐색은 4회에서 6회로 증가합니다.

순차 탐색 vs 이진 탐색 — 비교표

원소 수 n	순차 탐색 최대 검사 횟수	이진 탐색 최대 중앙 검사	검사 횟수 비율 순차 ÷ 이진
32	32회	**6회**	약 5배
1,000	1,000회	**10회**	약 100배
10,000	10,000회	**14회**	약 714배
1,000,000	1,000,000회	**20회**	**약 50,000배**
1,000,000,000	10억 회	**30회**	**약 3,300만 배**

SECTION 1.5

알고리즘 분석

1.5

실행 시간과 메모리 사용량을 어떻게 평가할까요? 기본 연산의 횟수와 시간 복잡도 함수 T(n)을 배웁니다.

알고리즘 분석이란?

알고리즘 분석이란, 주어진 알고리즘이 얼마나 효율적으로 동작하는지 평가하는 과정입니다.

실행 시간
얼마나 빠르게 동작하는가

메모리 사용량
얼마나 많은 공간을 쓰는가

시간 복잡도 분석이란?

실측 시간은 환경에 따라 달라집니다. 여기서는 기본 연산 횟수로 입력 크기에 따른 증가를 분석합니다.

관계없는 기준을 사용

"기본 연산(basic operation)이 몇 번 일어나는가?" 를 세어 효율성을 평가합니다.

시간 복잡도 함수 T(n)

기본 연산의 횟수는 보통 입력의 개수(크기) n에 따라 변합니다. 이를 n의 함수로 나타낸 것이 시간 복잡도 함수입니다.

T(n)

시간 복잡도 함수

정렬 · 탐색 리스트의 크기 n

다항식 계산 다항식의 차수

행렬 곱셈 행렬의 차원 n

문자열 처리 문자의 개수

입력 크기 n이 커질 때 T(n)이 어떻게 증가하는지가 알고리즘의 효율을 결정합니다.

기본 연산 (Basic Operation)

알고리즘이 하는 일 중에서 가장 핵심적인 명령 하나를 대표로 뽑은 것이 기본 연산입니다.

예: 탐색 알고리즘의 기본 연산 = 비교 연산
순차 탐색·이진 탐색 모두 찾는 값 x를 리스트 값과 하나씩(또는 절반씩) 비교하기 때문입니다.

비용 모형을 정한다
무엇을 한 번으로 세는지 명시하고 일관되게 비교합니다.

핵심적인 일을 고른다
대표 연산을 세되, 다른 연산이 전체 비용을 지배하는지도 확인합니다.

무엇을 한 번으로 셀 것인가

입력 크기, 기본 연산, 분석할 경우를 먼저 정합니다.

알고리즘	입력 크기	세는 단위	예
배열 합	원소 수 n	total+x 덧셈	n회
최댓값	원소 수 n	현재 값과 최댓값 비교	n−1회
이진 탐색	원소 수 n	중앙 원소 검사	최대 $\lfloor \log_2 n \rfloor + 1$회
GCD	정수의 크기/비트 수	나머지 계산 등	모형을 명시

기본 연산 횟수 T(n)과 실제 실행 시간(초)은 구별합니다.

시간 복잡도와 보조 공간 복잡도

입력을 저장한 공간을 제외하고, 추가로 필요한 메모리를 비교합니다.

작업	추가로 저장하는 것	보조 공간
배열 합	합계와 반복 변수	Θ(1)
배열 복사	n개 원소를 담는 새 배열	Θ(n)
반복형 이진 탐색	low, high, mid	Θ(1)
재귀형 이진 탐색	최악 로그 깊이의 호출 스택	Θ(log n)

재귀형은 경계 인덱스를 전달하고 배열을 복사하지 않는 구현을 가정합니다.

예제 1.4

최대값 찾기 — 기본 연산의 개수

get_largest.py

```python
# 리스트 A 안에서 가장 큰 값을 찾는다.
def get_largest(A):
    largest = A[0]                  # 초기 최대값 설정
    for i in range(1, len(A)):      # 두 번째 원소부터 검사
        if A[i] > largest:          # 기본 연산: 비교
            largest = A[i]          # 최대값 갱신
    return largest
```

기본 연산

A[i] > largest

총

(n − 1)회

최댓값을 찾는 데 n−1회가 필요한 이유

비교만으로 서로 다른 n개 값의 최댓값을 찾는 모형입니다.

01 처음에는 n개 값 모두가 최댓값 후보입니다.

02 두 후보를 비교하면 작은 값 하나를 후보에서 제외할 수 있습니다.

03 후보 하나만 남기려면 n−1개를 제외해야 합니다.

04 따라서 적어도 n−1회 비교가 필요합니다.
앞의 알고리즘은 정확히 n−1회 비교하므로 이 기준에서 최적입니다.

예제 1.5

배열 요소의 합 — 시간 복잡도

sum_array.py

```python
# 배열의 모든 원소를 더하여 합계를 구한다.
def add_array_members(A):
    total = 0
    for x in A:           # n번 순회
        total = total + x   # 기본 연산: 덧셈 1회
    return total
```

반복문은 정확히

n번

실행되므로

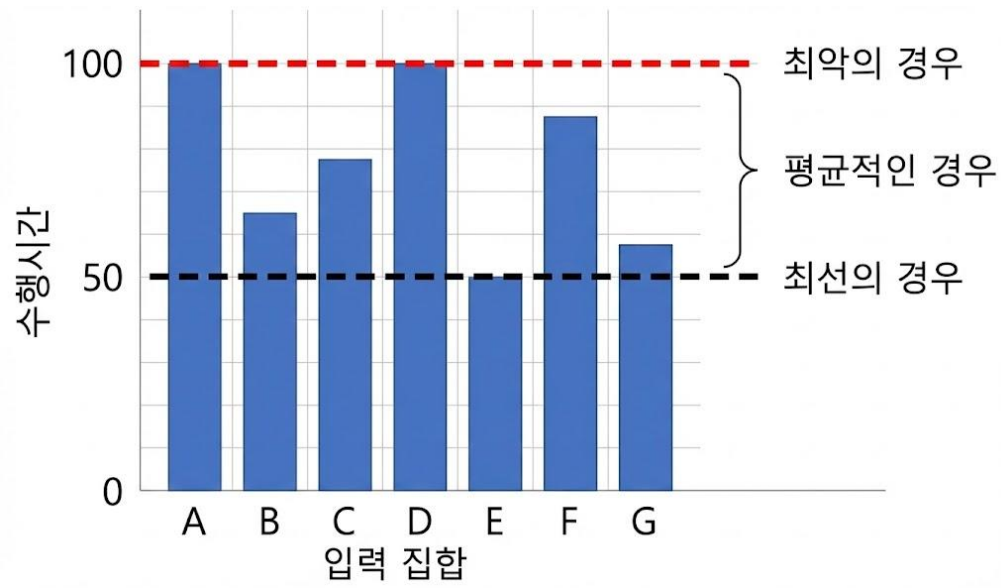
T(n) = n

선형적으로 증가

최선, 평균, 최악의 경우

똑같은 알고리즘도 주어지는 입력에 따라 다른 수행 시간을 보일 수 있습니다.

예: 순차 탐색
찾는 값이 리스트 맨 처음에 있으면 한 번에 끝나지만 , 맨 끝에 있으면 끝까지 가야 발견할 수 있습니다.



최선 · 평균 · 최악의 경우

최악의 경우

Worst Case

입력이 가장 불리할 때 실행 시간이 가장 길다.

찾는 값이 맨 마지막에 있는 경우

최선의 경우

Best Case

입력이 가장 유리할 때 실행 시간이 가장 짧다.

찾는 값이 맨 처음에 있는 경우

평균적인 경우

Average Case

여러 입력을 고려한 평균 실행 시간.

실용적인 성능 평가 기준으로 사용

비유: 엘리베이터 기다리기

최선
문이 열리자마자 바로 탑승

평균
평균은 운행 확률에 따라 달라짐

최악
꼭대기부터 전 층을 다 거침

예제 1.6

순차 탐색의 최악 · 최선 · 평균

linear_search.py

```python
def linear_search(S, target):
    for i in range(len(S)):      # 처음부터 끝까지 탐색
        if S[i] == target:       # 값이 일치하면
            return i             # 위치 반환
    return -1                    # 못 찾으면 -1
```

최선	1회
평균	(n+1)/2회*
최악	n회

*평균의 가정: 서로 다른 값, 탐색 성공, 각 위치의 확률이 동일함.
실패하면 n회 검사합니다. 자세한 평균 계산은 다음 슬라이드에서 풉니다.

평균은 확률을 정해야 계산할 수 있습니다

서로 다른 값, 탐색 성공, 각 위치가 같은 확률이라는 가정을 둡니다.

찾는 위치	첫 번째	두 번째	세 번째	네 번째
확률	1/4	1/4	1/4	1/4
비교 횟수	1	2	3	4

평균 = (1+2+3+4) / 4 = 2.5회

일반적으로 (1+2+…+n)/n = (n+1)/2회입니다.

어떤 복잡도를 사용할 것인가?

최악은 모든 입력에 대한 상한을, 평균은 가정한 확률 분포에서의 기댓값을 설명합니다. 최선도 입력에 따른 차이를 이해하는 데 유용합니다.

안전한 설계 → 최악의 경우
항상 최악에서도 문제없이 작동해야 할 때

실용적 평가 → 평균의 경우
일반적인 상황의 성능을 볼 때

예제 1.7

순차 탐색의 평균적인 경우 계산

가정 target이 리스트에 존재하고, 모든 항목이 서로 다르며, 어느 위치에 있을 확률도 모두 동일하다. (위치 k에 있을 확률 = 1/n)

① target이 리스트에 존재할 때 — 평균 비교 횟수

$$T_{avg}(n) = \frac{1+2+3...+n}{n} = \frac{n+1}{2}$$

② 존재 확률이 p일 때 — 존재하지 않는 경우까지 포함

$$T_{avg}(n) = \frac{p}{n}(1+2+3...+n)+(1-p)n = n(1-\frac{p}{2})+\frac{p}{2}$$

1부터 n까지의 합 증명

앞에서 더한 합과 뒤에서 더한 합을 세로로 짝지으면 매번 n+1이 됩니다.

$$S = 1 + 2 + \cdots + (n-1) + n$$
$$S = n + (n-1) + \cdots + 2 + 1$$

$2S = n(n+1)$
$S = n(n+1)/2$

평균 비교 횟수 = S/n = (n+1)/2입니다.

실패하는 탐색까지 포함한 평균

성공 확률을 p, 성공한 경우 각 위치의 확률을 동일하게 가정합니다.

$$E[C] = p \times (n+1)/2 + (1-p) \times n$$

n = 10	성공 시 평균	실패 시 횟수	전체 평균
p = 1	5.5	10	5.5
p = 0.8	**5.5**	**10**	**6.4**
p = 0	5.5	10	10

p=0.8일 때: 0.8×5.5 + 0.2×10 = 6.4회

SECTION 1.6

점근표기법 (Big-O)

1.6

입력이 커질 때 실행 시간이 어떻게 증가하는지, 그 성장률을 O · Ω · Θ 로 표현합니다.

최고차항이 성장률을 결정하는 이유

$T(n)=n^2+n+1$에서 작은 항의 비중이 어떻게 줄어드는지 계산합니다.

n	n^2	n+1	n^2의 비중
10	100	11	**약 90.09%**
100	10,000	101	**약 99.00%**
1,000	1,000,000	1,001	**약 99.90%**

$T(n)/n^2 = 1 + 1/n + 1/n^2$: n이 커질수록 이 비율은 1에 가까워집니다.

세 가지 점근 표기법

표기법	의미	알고리즘 관점	비유
빅오 **Big-O**	상한 충분히 큰 n에서 고정 상수 배 이하	이 함수보다 빠르게 증가하지 않는다	"최대 이 정도까지 걸린다"
빅오메가 **Big-Ω**	하한 충분히 큰 n에서 고정 상수 배 이상	이 함수보다 느리게 증가하지 않는다	"적어도 이 정도는 걸린다"
빅세타 **Big-Θ**	같은 성장률의 상·하한 두 고정 상수 배 사이	정확한 성장률을 표현한다	"이 정도 크기로 증가한다"

빅오 표기법 (Big-O)

빅오 표기법은 연산의 횟수를 대략적(점근적)으로 표기한 것으로, 함수의 상한을 나타냅니다.

정의

두 함수 f(n), g(n)에 대해 모든 $n \geq n_0$에서

$|f(n)| \leq c \cdot |g(n)|$

을 만족하는 양의 상수 c, n_0가 존재하면

$f(n) = O(g(n))$

예시

$n \geq 5$ 이면

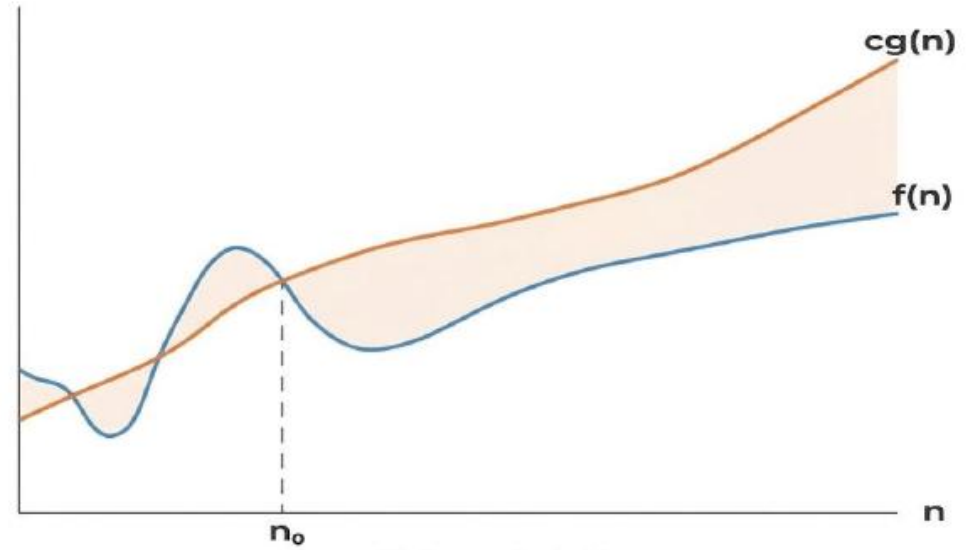
$2n + 1 < 10n$

이므로

$2n + 1 = O(n)$

핵심 빅오는 "이 함수보다 빠르게 증가하지 않는다" 는 상한을 표시합니다. 실무에서는 최소 차수로 상한을 표현합니다.

빅오 표기법 — 상한 (Upper Bound)



f(n) = O(g(n))

그림 1.7 빅오 표기법

$n \geq n_0$ 이후
$c \cdot g(n)$이 항상 f(n)
보다 위에 있습니다.

즉 $c \cdot g(n)$이 상한이므로 f(n) = O(g(n))

Big-O 증명에서 c와 n_0 고르기

f(n)=2n+1이 O(n)임을 보이려면 고정 상수 c와 n_0를 찾습니다.

01 목표 부등식: $2n+1 \le c \cdot n$

02 $n \ge 1$이면 $1 \le n$이므로 $2n+1 \le 2n+n = 3n$입니다.

03 $c=3$, $n_0=1$을 선택하면 모든 $n \ge n_0$에서 성립합니다.

04 따라서 $2n+1 = O(n)$입니다.
상수 c와 시작점 n_0는 유일하지 않습니다.

다항식의 $\Theta(n^2)$ 증명

$f(n)=3n^2+5n+2$를 n^2의 두 상수 배 사이에 가둡니다.

$$3n^2 \le 3n^2+5n+2 \le 10n^2$$

01 하한: 양수 항을 더했으므로 $f(n) \ge 3n^2$입니다.

02 상한: $n \ge 1$이면 $5n \le 5n^2$, $2 \le 2n^2$입니다.

03 $c_1=3$, $c_2=10$, $n_0=1$이므로 $f(n)=\Theta(n^2)$입니다.

n^2은 왜 O(n)이 아닐까?

어떤 고정 상수 c를 골라도 n^2을 $c \cdot n$ 아래에 계속 둘 수 없습니다.

01 $n^2 = O(n)$이라고 가정하면 $n^2 \leq c \cdot n$이어야 합니다.

02 $n>0$으로 나누면 $n \leq c$가 됩니다.

03 하지만 n은 계속 커집니다. $n \geq n_0$이면서 $n>c$인 수를 고를 수 있습니다.

04 가정과 모순이므로 n^2은 O(n)이 아닙니다.
반대로 $n \leq n^2$이므로 $n = O(n^2)$은 맞습니다.

대표 복잡도와 적용 조건

정확한 차수를 말할 때는 Θ, 상한만 말할 때는 O를 사용합니다.

성장률	대표 예	확인할 조건
$\Theta(1)$	배열 인덱스 접근	고정 크기 원소
$\Theta(\log n)$	이진 탐색의 최악	정렬된 배열
$\Theta(n)$	배열 합	n개 원소 처리
$\Theta(n \log n)$	병합 정렬의 최악	일반적인 비교 기반 구현
$\Theta(n^2)$	앞의 버블 정렬 비교 수	조기 종료 최적화 없음
$O(2^n)$	단순 재귀 피보나치	메모이제이션 없음

퀵 정렬: 기대 $\Theta(n \log n)$, 최악 $\Theta(n^2)$. 순열은 경우의 수만 $n!$개입니다.

병합 정렬에서 $n \log_2 n$이 나오는 이유

$n=2^k$인 경우를 생각하면 분할 깊이는 $\log_2 n$이고, 각 병합 수준의 작업량은 $\Theta(n)$입니다.

n=8의 병합 단계	병합 작업	처리하는 원소 수 합
첫 수준	길이 2를 4번	**2×4 = 8**
둘째 수준	길이 4를 2번	**4×2 = 8**
마지막 수준	길이 8을 1번	**8×1 = 8**

각 수준 $\Theta(n)$ × 병합 수준 $\log_2 n$ = $\Theta(n \log n)$

버블 정렬의 한 회전

[4, 1, 3, 2]에서 인접한 두 수를 비교하며 큰 수를 오른쪽으로 보냅니다.

단계	비교한 값	교환 후 배열
시작	없음	[4, 1, 3, 2]
1	4와 1	[1, 4, 3, 2]
2	4와 3	[1, 3, 4, 2]
3	4와 2	**[1, 3, 2, 4]**

첫 회전이 끝나면 최댓값 4의 위치가 확정됩니다. 다음 회전은 앞의 3개만 봅니다.

버블 정렬의 정확한 비교 횟수

안쪽 반복 횟수가 회전마다 달라지므로 합으로 계산합니다.

```
for i in range(n - 1):
    for j in range(n - i - 1):
        if A[j] > A[j + 1]:
            A[j], A[j+1] = A[j+1], A[j]
```

$$T(n) = (n-1)+(n-2)+\cdots+1$$
$$= n(n-1)/2 = \Theta(n^2)$$

n=4이면 3+2+1=6회 비교합니다. 단순히 두 반복문의 횟수를 곱하지 않습니다.

이진 탐색의 시간 복잡도 정리

한 반복의 비용은 상수이고, 최악의 반복 횟수는 $\lfloor \log_2 n \rfloor + 1$입니다.

01 범위의 크기는 매번 적어도 절반 수준으로 줄어듭니다.

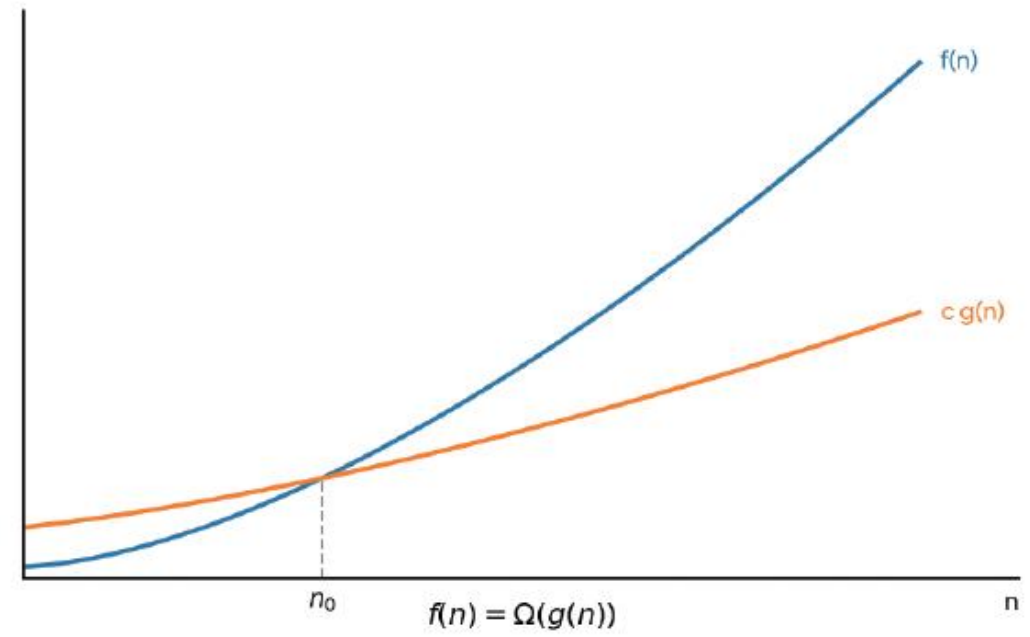
02 최악 반복 횟수 $C(n) = \lfloor \log_2 n \rfloor + 1$입니다.

03 한 반복의 비용이 $\Theta(1)$이므로 최악 시간은 $\Theta(\log n)$입니다.

04 첫 중앙값이 정답이면 최선 시간은 $\Theta(1)$입니다.

$O(\log n)$은 올바른 상한입니다. $\Theta(\log n)$은 최악 성장률을 더 정확하게 말합니다.

빅오메가 표기법 (Big-Ω)



정의

모든 $n \geq n_0$ 에서 $|f(n)| \geq c \cdot |g(n)|$
을 만족하는 c, n_0가 존재하면
$f(n) = \Omega(g(n))$

함수의 하한 (Lower Bound)
"이 함수보다 느리게 증가하지 않는다" 는 뜻입니다.

예) $n = \Omega(n)$

빅세타 표기법 (Big-Θ)

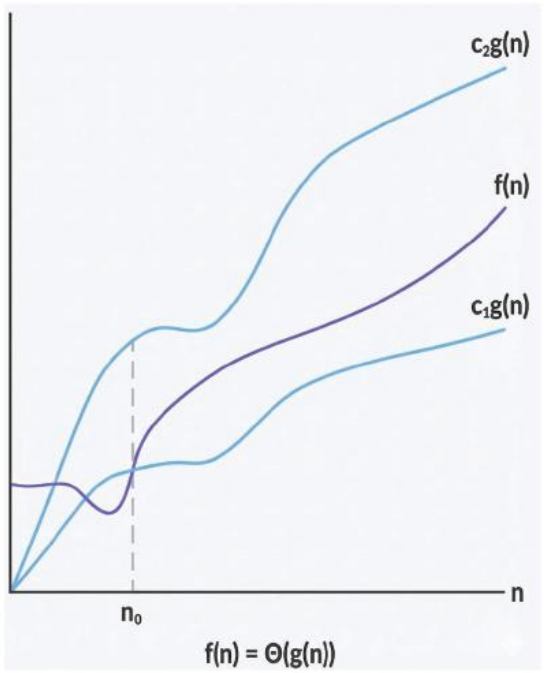


그림 1.8 빅세타 표기법

정의

모든 $n \geq n_0$ 에서

$c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|$

를 만족하는 c_1, c_2, n_0가 존재하면

$f(n) = \Theta(g(n))$

하한인 동시에 상한 (Tight Bound)

$f(n) = O(g(n))$ 이면서 $f(n) = \Omega(g(n))$ 이면 $f(n) = \Theta(g(n))$

예) $n \geq 1$ 이면 $n \leq 2n+1 \leq 3n$ 이므로 $2n+1 = \Theta(n)$

O는 상한, 최악은 입력의 선택

최선·평균·최악과 O·Ω·Θ는 서로 다른 분류입니다.

순차 탐색의 경우	비교 횟수 함수	차수
최선	1	Θ(1)
성공 위치가 균등한 평균	(n+1)/2	Θ(n)
최악	n	Θ(n)

최악 함수도 Ω(n)과 Θ(n)으로 표현할 수 있습니다. O = 최악이라는 뜻은 아닙니다.

입력이 커질 때의 대표 성장률

아래 순서는 충분히 큰 n에서의 성장률 비교입니다.

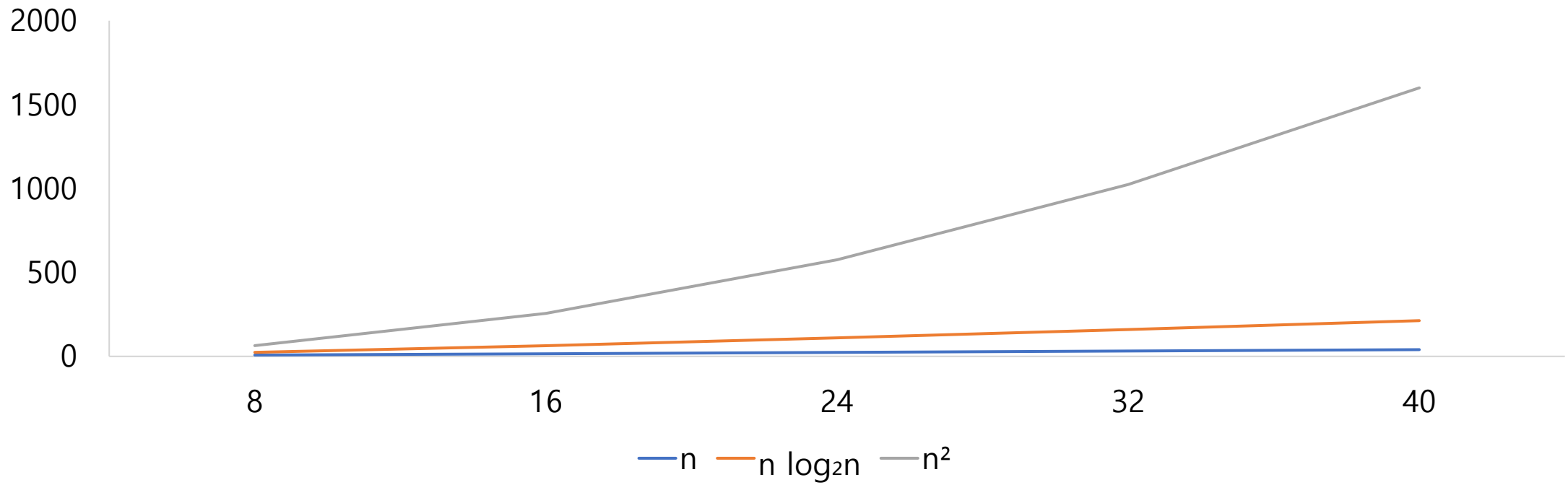
$$1 < \log_2 n < n < n \log_2 n < n^2$$

$$< n^3 < 2^n < n!$$

같은 문제, 같은 입력 조건, 같은 비용 모형에서 알고리즘을 비교해야 합니다.

선형·선형로그·이차 함수의 그래프

표시한 입력 크기에서 직접 계산한 함수값입니다. 시간 단위는 아닙니다.



n=32에서 n은 32, n log₂n은 160, n²은 1,024입니다.

복잡도 함수의 수치 비교

로그의 밑은 2입니다. 모두 대표 함수의 값이며 실측 시간은 아닙니다.

입력 n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
4	2	4	8	16	16
8	3	8	24	64	256
16	4	16	64	256	65,536
32	5	32	160	1,024	4,294,967,296

n을 두 배로 늘리면: $\log_2 n$은 1 증가, n은 2배, n^2은 4배가 됩니다.

반복문 복잡도 연습

각 코드에서 work() 실행 횟수를 먼저 세고, 그다음 Θ로 표현해 봅니다.

A

```
for i in range(n):
    work()
```

B

```
for i in range(n):
    for j in range(n):
        work()
```

C

```
for i in range(n):
    j = 1
    while j < n:
        work()
        j *= 2
```

가정: n≥2인 정수, work() 한 번의 비용은 Θ(1)입니다.

반복문 연습 풀이

반복문이 두 겹이어도 무조건 $\Theta(n^2)$은 아닙니다.

문항	work() 실행 횟수	복잡도
A	n	$\Theta(n)$
B	$n \times n$	$\Theta(n^2)$
C	$n \times \lceil \log_2 n \rceil$	$\Theta(n \log n)$

버블 정렬처럼 안쪽 범위가 달라지면 각 회전의 횟수를 합산합니다.

로그와 증명 확인 문제

계산 결과와 함께 "왜 그런가"를 한 문장으로 설명해 봅니다.

01 $\log_2 64$는 얼마인가요? 2의 거듭제곱 식으로 바꿔 보세요.

02 정렬된 64개 원소의 이진 탐색은 최악에 몇 번 검사하나요?

03 $5n+7 = O(n)$을 보이는 c, n_0 한 쌍을 제시해 보세요.

04 "반복하면 후보 구간이 줄어든다"만으로
이진 탐색의 정답까지 보장할 수 있나요?

로그와 증명 확인 문제 풀이

정확한 횟수, 성장률, 정확성의 근거를 구별합니다.

01 $\log_2 64 = 6$입니다. $2^6 = 64$이기 때문입니다.

02 $\lfloor \log_2 64 \rfloor + 1 = 7$회입니다. 마지막 원소도 확인합니다.

03 $n \geq 1$이면 $5n+7 \leq 12n$입니다. $c=12$, $n_0=1$이면 됩니다.

04 구간 감소는 종료를 보장합니다. 정답을 보장하려면 "존재하는 정답을 구간 안에 유지한다"는 불변식도 필요합니다.